A Real-Time Procedural Shading System for Programmable Graphics Hardware

Kekoa Proudfoot* Stanford University
William R. Mark* Stanford University
Svetoslav Tzvetkov† NVIDIA Corporation
Pat Hanrahan* Stanford University

Abstract

Real-time graphics hardware is becoming programmable, but this programmable hardware is complex and difficult to use given current APIs. Higher-level abstractions would both increase programmer productivity and make programs more portable. However, it is challenging to raise the abstraction level while still providing high performance. We have developed a real-time procedural shading language system designed to achieve this goal.

Our system is organized around multiple *computation frequencies*. For example, computations may be associated with vertices or with fragments/pixels. Our system's shading language provides a unified interface that allows a single procedure to include operations from more than one computation frequency.

Internally, our system virtualizes limited hardware resources to allow for arbitrarily-complex computations. We map operations to graphics hardware if possible, or to the host CPU as a last resort. This mapping is performed by compiler back-end modules associated with each computation frequency. Our system can map vertex operations to either programmable vertex hardware or to the host CPU, and can map fragment operations to either programmable fragment hardware or to multipass OpenGL. By carefully designing all the components of the system, we are able to generate highly-optimized code. We demonstrate our system running in real-time on a variety of hardware.

CR Categories: I.3.3 [Computer Graphics]: Picture/Image Generation; I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Color, shading, shadowing, and texture; D.3.2 [Programming Languages]: Language Classifications—Specialized application languages; I.3.1 [Computer Graphics]: Hardware Architectures—Graphics processors

Keywords: graphics hardware, graphics systems, shading languages, rendering

1 Introduction

Mainstream graphics hardware is rapidly moving toward programmability. An important first step was the addition of new features such as multitexturing, configurable texture blending units, and per-fragment dot products. The latest generation of hardware, as described in [10, 14, 16], directly supports programmable vertex and fragment computations. This hardware enables a large number of interesting new special effects for games and other interactive applications.

While the latest hardware features are very flexible, the same hardware features are difficult to use. This is true for two reasons. First, current hardware interfaces are low-level. Programmability is exposed through the graphics library, either through an assembly-language-like interface to functional units or through an explicit pipeline-configuration model. Second, since hardware and extensions vary across vendors and product generations, writing efficient portable software is challenging, and often requires customizing applications to each supported platform. These two problems decrease programmer productivity and make it harder for vendors to convince users to adopt new features.

To fix the ease-of-use problem, new programming models and higher-level hardware abstractions are needed. Higher-level abstractions can provide standard programmable interfaces that both simplify underlying complexities and hide differences across implementations. Shading languages have evolved to solve the abstraction problem for software rendering systems, and we believe that shading languages are appropriate for abstracting graphics hardware.

In this paper, we describe our procedural shading system. We make three contributions. First, we develop and describe a new programmable pipeline abstraction that combines and extends elements from previous work. Second, we describe a new shading language with features appropriate to our abstraction and to current and future hardware. Third, we describe a retargetable compiler back end that maps our abstraction to a variety of different graphics accelerators, including those with vertex and fragment programmability, using a set of interchangeable compiler modules.

The resulting system makes hardware much easier to program by efficiently mapping a shading language to the wide variety of hardware available today. We show vertex and fragment back ends that target programmable graphics hardware efficiently, and we demonstrate several complex scenes with programmed shaders running in real-time on PC graphics hardware.

2 Background

Shading languages developed from the work of Cook, who described how shade trees could provide a flexible, programmable framework for shading computations [3], and the work of Perlin, who described a how a language could be used for processing pixel streams [20]. The most common shading language in use today is the RenderMan Shading Language [1, 5, 23], which provides for movie production-quality procedural shading effects for software batch-rendering systems.

More recently, several systems have demonstrated shading languages targeted to real-time rendering and graphics hardware.

Olano and Lastra [18] describe pfman, a RenderMan-like language for the PixelFlow system [15] that is compiled to PixelFlow's SIMD processing arrays. While PixelFlow is well-suited to programmable shading, for many reasons, today's mainstream hardware bears little resemblance to it.

*{kekoa, billmark, hanrahan}@graphics.stanford.edu
†svetkins@nvidia.com; previously at Stanford University

id Software's Quake III Arena includes a pass-based scripting language that allows multiple layers of textures and colors to be animated and composited using basic hardware blending [7]. The graphics engine maps passes in the language to actual hardware passes, using multitexture to compress passes when possible. The language also contains mechanisms for generating and manipulating texture coordinates.

Peercy et al. describe an approach to implementing shading languages using multipass rendering [19]. They showed that the RenderMan Shading Language could be compiled using multipass rendering given OpenGL 1.2 with the imaging subset plus two hardware extensions: support for extended range and precision (e.g. their 16-bit floating-point representation) and dependent texturing. For hardware without these extensions, they developed a simpler language, called ISL, that exposes functionality available in OpenGL 1.2 and provides a convenient way to describe basic computations involving colors, textures, and the output of the configurable OpenGL vertex-based lighting model.

The key insight behind the Peercy et al. approach is to abstract the graphics pipeline as a SIMD processor. Each configuration of the OpenGL graphics pipeline corresponds to a different SIMD *instruction*. One pass then represents the execution of one such instruction. The SIMD nature of the processor arises because each rendering pass performs the same operation across many fragments. The SIMD processor model provides a framework for abstracting multipass rendering, which in turn allows the model to express arbitrarily-complex fragment computations.

Recent hardware exposes a different kind of hardware abstraction, namely the programmable vertex/fragment-processing model of DirectX 8 [14] and NVIDIA's NV_vertex_program and NV_register_combiner [16] OpenGL extensions. This model replaces portions of the traditional non-programmable rendering pipeline with programmable register-machine-based processing units. Rather than exposing programmability as multiple rendering passes, this hardware model places programmability entirely within a single rendering pass. As a result, the programmable model treats a pass as a series of *many simple instructions*, unlike the SIMD processor model which treats a pass as a single complex instruction. For vertex processing, DirectX 8 and NVIDIA both provide a set of floating-point operations sufficient for implementing standard transform and lighting calculations. For fragment processing, DirectX 8 supports a set of standard texture combining operations as instructions, with a limit of eight instructions per pass, while NVIDIA's register combiners expose similar functionality, except the combining operations are more powerful and more complex.

There are two major differences between the programmable vertex/fragment-processing model and the fragment-centric SIMD processor model.

The first difference is availability of programmable vertex processing. In the near-term, support for programmable vertex processing provides two advantages:

- Vertex-processing hardware provides many useful operations not present in current fragment-processing hardware, such as division and square root.
- Vertex-processing hardware provides floating-point arithmetic, while current fragment-processing hardware is limited to 8- or 9-bit signed arithmetic for most operations. Low-precision fixed-point arithmetic is insufficient for many computations and motivated Peercy's proposal for extended-precision support [19].

While we expect fragment hardware to eventually catch up to vertex hardware, vertex programmability allows users to implement many computations not possible using today's fragment hardware.

More fundamentally though,

- Vertex programmability provides a natural and efficient way to perform position, texture coordinate, and lighting computations because these quantities often vary slowly across a surface. Furthermore, the ability to perform computations for each vertex maps well to programmers' conceptual model of the graphics pipeline.

A second difference between the programmable processing model and the SIMD processor model lies in how well the two models abstract hardware that can perform many operations per pass:

- In high-performance graphics systems, bandwidth between the graphics chip and framebuffer is scarce, as is bandwidth between the graphics chip and host. Each rendering pass consumes bandwidth, so it is important to minimize the number of rendering passes. Moreover, because of VLSI technology trends, the ratio of graphics chip compute performance to external memory bandwidth will continue to increase. This trend favors designs where more operations are done per pass.
- As more operations are performed in each pass, the SIMD processor model of a pass as a single complex instruction breaks down. Although instructions could be designed to contain more operations, there is little agreement on which combinations of operations are needed. On the other hand, a programmable processing model that provides for many simpler instructions in a single pass naturally and effectively abstracts increasing numbers of operations per pass.

We believe these two differences make the programmable processing model a better abstraction for future hardware. Therefore, we base our system on that model. However, we extend the approach taken by Microsoft and NVIDIA in three important ways:

- We generalize vertex/fragment processing using the concept of multiple computation frequencies to allow for operations at rates other than simply per-vertex or per-fragment.
- We provide a single unified framework, called the *programmable pipeline*, that combines all computation frequencies into a single abstraction to simplify the process of specifying shading computations.
- We virtualize the existing hardware-centric pipelines to remove resource constraints. Programmers need not be aware of physical resource limits such as the number of internal registers and the number of instructions. One method of virtualization is to apply multipass methods similar to those used by the SIMD processor model.

McCool [13] recently proposed the SMASH API. SMASH advocates programming hardware using a stack-machine-based API for specifying operations, and shows examples of several different metaprogramming techniques for exposing this functionality to application developers. In contrast, we focus on the shading language as the primary interface. Despite this difference, the underlying capabilities exposed by SMASH are similar to those of our programmable pipeline. Another difference between the SMASH processing model and ours is that SMASH assumes vertex programmability is post-transform (to alleviate the need for common-case transform code) and post-backface-cull (to eliminate vertex computations for culled vertices). These assumptions imply that position computations are not programmable. Our system allows position computations to be programmed, although our language does not yet provide direct access to this feature. A final difference is we do not focus on the details of the hardware interface. Instead, we examine specific language features and associated compiler analysis and optimization techniques, and we develop a variety of retargetable compiler back ends that target a number of different platforms.

Olano [17] describes a programmable pipeline for graphics hardware. His programmable pipeline contains programmable stages corresponding to transformation, rasterization, interpolation, shading, lighting, etc. Programmability is implemented using

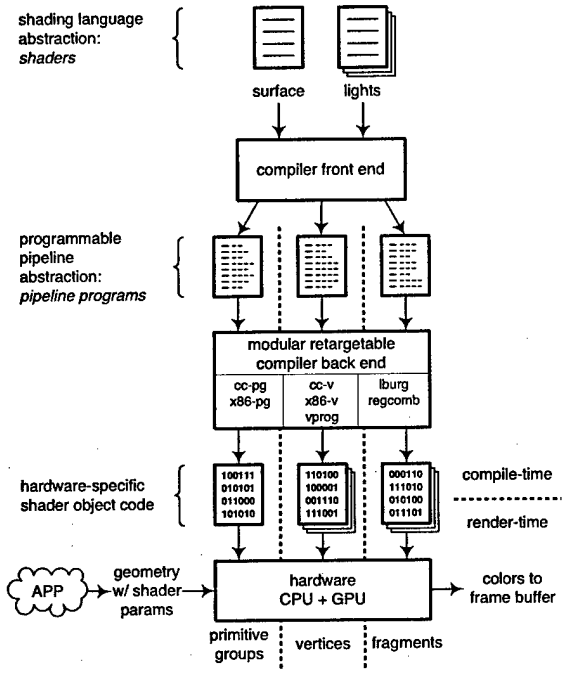


Figure 1: System block diagram. In our system, surface and light shaders are compiled into three-part pipeline programs split by computation frequency. For each computation frequency, we apply a back-end module to generate shader object code that is executed during rendering. We implement seven modules; most notably, we implement a pair of modules to target vertex programs and register combiners.

PixelFlow's SIMD processing arrays, and shading and lighting are always computed per-fragment. Because his stages are organized by function as opposed to computation frequency, his notion of a programmable pipeline is different from ours.

3 System Overview

A block diagram of our system is shown in Figure 1. The principal components of our system are:

- **Shading language and compiler front end.** Shaders in our shading language are used to describe shading computations. A compiler front end maps the shading language to an intermediate pipeline program representation.
- **Programmable pipeline abstraction.** An intermediate abstraction layer provides a generic interface to hardware programmability to hide hardware details and to simplify compiler front ends. It consists of a computational model (the programmable pipeline) and a means for specifying computations (pipeline programs). Pipeline programs are divided into pieces by computation frequency.
- **Retargetable compiler back end.** A modular, retargetable compiler back end maps pipeline programs to shader object code. There are back-end modules for different stages and for different hardware.
- **Shader object code.** Compiled shaders are used to configure hardware during rendering. Shader object code separates the compile-time and render-time halves of our system.
- **Shader execution engine.** A shader execution engine controls the rendering of geometric primitives using the shader object code. The application may attach shader parameters to groups of primitives and to vertices. These parameters are processed to compute surface positions and surface colors.
- **Graphics hardware.** Shader execution modules rely on graphics hardware for most shading computations, although the host CPU may be used for some computations.

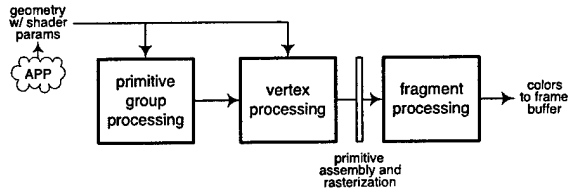


Figure 2: Programmable pipeline abstraction. The programmable pipeline is an abstraction layer consisting of three programmable stages, one for each of three computation frequencies. Stages execute a pipeline program (not shown) to process geometric primitives with associated shader parameters. The results of each stage are passed to subsequent stages. Between programmable stages are fixed-function stages that convert values between computation frequencies.

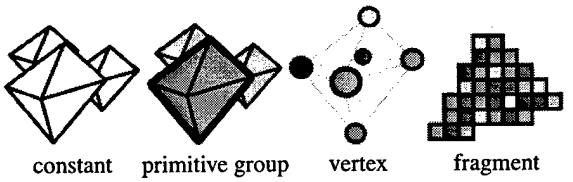


Figure 3: Computation frequencies. Our system supports four computation frequencies. In the illustrations above, individual elements at each computation frequency are depicted by color.

Our system runs on top of OpenGL. Like OpenGL, our system renders objects sequentially. After specifying and compiling a number of shaders, the user repeatedly alternates between binding shaders and specifying objects for rendering. Objects are drawn immediately after they are specified.

Our prototype provides both immediate-mode and vertex array interfaces. These interfaces rely on buffered data to automatically handle multiple rendering passes. While geometry specified using the vertex array interface is inherently buffered, geometry specified using the immediate-mode interface is buffered into vertex arrays on the fly. Vertex arrays are then passed to the shader execution engine.

4 Programmable Pipeline Abstraction

The programmable pipeline abstraction is the central element of our shading system. It provides an abstraction that simplifies mapping our shading language to hardware. It provides a computation model that describes what and how different values are computed. It also defines how computations are expressed and which operators may be used to perform computations. In this section, we describe the key elements of the programmable pipeline abstraction.

4.1 Pipeline operation

Our programmable pipeline abstraction is illustrated in Figure 2. The programmable pipeline renders objects by computing positions and colors. Computed positions are used to control rasterization and depth buffering, while computed colors are blended into the framebuffer. The abstraction contains two kinds of stages: programmable stages and fixed-function stages.

Programmable stages are associated with different computation frequencies. We support four computation frequencies: constant, per-primitive-group, per-vertex, and per-fragment. We illustrate these computation frequencies in Figure 3. Constant expressions are evaluated once at compile time and are not evaluated by the programmable pipeline. Primitive groups are defined as the geometry within an OpenGL Begin/End pair; vertices are defined by the OpenGL Vertex command; and fragments are defined by the screen-space sampling grid. (In this context, a primitive is a single point, line, or polygon; in general, a Begin/End pair can

specify a group of such primitives.) On today's hardware, multiple computation frequencies enable a tradeoff between complex high-precision floating-point computations at a coarse level of detail and many simple low-precision fixed-point computations at a fine level of detail.

Programmable stages are ordered by frequency, least-frequent first. Each stage processes a stream of independent objects (e.g. individual vertices or fragments). Stages compute an output stream given an input stream composed of application-specified parameters and outputs from previous pipeline stages.

Between consecutive programmable stages are fixed-function stages that implement the parts of the graphics pipeline that cannot be programmed. In particular, between the programmable vertex and fragment stages are stages that assemble vertices into primitives and rasterize primitives to fragments. The rasterization stage also interpolates vertex values such as texture coordinates and colors to obtain fragment values.

Programmable stages are driven by a pipeline program consisting of operators arranged into directed acyclic graphs (DAGs). The DAGs are partitioned by pipeline stage, and specify how to compute stage outputs from stage inputs.

Pipeline programs virtualize the hardware. There are no program size limits and no limits to the number of inputs, outputs, and parameters allowed. Conceptually the programmable pipeline performs all computations in a single pass. In practice, however, large computations may be split into multiple passes. Our abstraction hides this splitting process.

4.2 Data types

Stages in our system operate on ten data types. There are scalars, 3-vectors, and 4-vectors, each of which may be composed of either floats or [0,1]-clamped floats. The remaining four types are 3×3 floating-point matrices, 4×4 floating-point matrices, booleans, and a special texture reference type that allows access to textures through the OpenGL texture naming mechanism.

All of our data types are abstract types: each type has a well-defined semantics but not necessarily a well-defined storage representation. For example, the floating-point type need not be represented as IEEE floating-point numbers. This allows us to easily map types to a wide variety of hardware, and follows principles established in the OpenGL specification [21].

The [0,1]-clamped float type is included to represent fixed point numbers, particularly fragment color values, as well as clamped floating-point values at vertices (normally vertex colors). Recent fragment-processing hardware supports larger fixed-point ranges (especially [–1,1]), but for reasons discussed in Section 6.5, we do not provide a [–1,1]-clamped data type at this time.

Although current fragment hardware does not support programmable floating-point computations, we provide a fragment floating-point type abstraction. This allows users to easily write surface shaders that can be used with either vertex or fragment lights. We implement the fragment float type using the best-available fragment data type. Since current fragment-processing hardware is implemented using fixed-point and therefore has limited range, overflows and clamping are possible. We expect this problem to go away in the future once fragment hardware supports a true floating-point type.

Our use of a clamped float type differs slightly from the works of Olano and McCool. Olano's language allows for well-defined fixed-point types specified with a given size and exponent (e.g. fixed<16,16>) [17]. This capability matches PixelFlow hardware, but not the graphics hardware we target. McCool provides a hinting mechanism for storage representations [13].

4.3 Operators

The operators we implement were chosen to support standard transform, lighting, and texturing operations. We purposely omit operations that cannot be implemented today.

We include support for: basic arithmetic; scalar, vector, and matrix manipulation; type casting; exponentiation, square roots, dot and cross products, trigonometric functions, and vector normalization; transformation matrix generation; comparisons; selection of one of two values based on third boolean value; min, max, and clamp operators; access to parameters and constants.

Several operations support texture lookups, including support for 2D textures, 3D textures, and cubical environment maps. All of the texture operations use textures specified outside our shading system through the OpenGL texture naming mechanism.

Certain complex operations can be difficult or impossible to express efficiently across a variety of hardware in terms of the other operators that are available. To make these operations efficient, we support a few high-level operators called *canned functions*. In particular, we include two canned functions (bumpspec and bumpdiff) to make bump mapping more efficient for one of our fragment back ends. These functions implement bump mapping as described by Kilgard [8].

Non-orthogonal operators. Ideally, every hardware platform would support every operator at every computation frequency; however, current hardware platforms are far from ideal. There are two kinds of operator non-orthogonalities: non-orthogonalities across computation frequencies and non-orthogonalities across hardware platforms. Examples of operators that are not orthogonal across computation frequencies include: divide and square root, which are not supported per-fragment; matrix generation functions, which are too expensive to implement more frequently than per-primitive-group; and texturing, which is fragment-only. Texturing is also limited because per-fragment texture coordinate computations are not fully supported. Examples of operators that are not orthogonal across platforms include cubemaps, bumpmaps, and 3D textures.

To guarantee that operators can be successfully mapped to hardware, we restrict the available set of operators to those the targeted hardware can actually implement. In recognition of the variety of hardware currently available and in anticipation of improved future hardware, we implemented a table-driven restriction mechanism. By storing restrictions in tables, we are able to specialize restrictions to the peculiarities of each hardware platform. The use of tables also simplifies the process of extending our system with new operators as they become available. The tables themselves indicate which operators are available and which computation frequencies they are available at. The tables also associate a range of computation frequencies with the inputs to each operator to allow us to e.g. restrict texture coordinates to vertex values on hardware that does not support dependent texturing.

Our shading language compiler can determine at run-time which operators are available. It uses this information to provide conditional-compilation directives to allow multiple code versions to be written.

Unsupported operations. We do not support meta-operations representing control structures, such as labels and branches. Although these kinds of operations are useful, they are not supported by current hardware pipelines. Also, aside from read-only texturing operations, we do not support generic random-access memory operations, such as pointer dereferencing. The reason for these restrictions is based on hardware design principles. Allowing branches and random memory accesses would significantly slow down highly pipelined, data-parallel graphics hardware.

The lack of label, branch, and random-access memory operations helps to simplify the analysis of pipeline programs. From a compilation standpoint, a pipeline program has one basic block and no pointer aliasing.

```
#include "lightmodels.h"
surface shader float4 bowling_pin (texref base, texref bruns, texref circle,
                                   texref coated, texref marks, float4 uv)
{
    // Compute per-vertex texture coordinates
    float4 uv_wrap = {uv[0], 10 * Pobj[1], 0, 1 };
    float4 uv_label = {10 * Pobj[0], 10 * Pobj[1], 0, 1};
    // Compute constant texture transformation matrices
    matrix4 m_base = invert(translate(0, -7.5, 0) * scale(0.667, 15, 1));
    matrix4 m_bruns = invert(translate(-2.6, -2.8, 0) * scale(5.2, 5.2, 1));
    matrix4 m_circle = invert(translate(-0.8, -1.15, 0) * scale(1.4, 1.4, 1));
    matrix4 m_coated = invert(translate(2.6, -2.8, 0) * scale(-5.2, 5.2, 1));
    matrix4 m_marks = invert(translate(2.0, 7.5, 0) * scale (4, -15, 1));
    // Compute per-vertex mask value to isolate front half of pin
    float front = select(Pobj[2] >= 0, 1, 0);
    // Transform texture coordinates, perform texture lookups, and apply mask
    float4 Base = texture(base, m_base * uv_wrap);
    float4 Bruns = front * texture(bruns, m_bruns * uv_label);
    float4 Circle = front * texture(circle, m_circle * uv_label);
    float4 Coated = (1 - front) * texture(coated, m_coated * uv_label);
    float4 Marks = texture(marks, m_marks * uv_wrap);
    // Invoke lighting models from lightmodels.h
    float4 Cd = lightmodel_diffuse({0.4, 0.4, 0.4, 1}, {0.5, 0.5, 0.5, 1});
    float4 Cs = lightmodel_specular({0.35, 0.35, 0.35, 1}, {0, 0, 0, 0}, 20);
    // Composite textures, apply lighting, and return final color
    return (Circle over (Bruns over (Coated over Base))) * Marks * Cd + Cs;
}
```

Figure 4: Example surface shader. This shader is adapted from the RenderMan bowling pin shader [23]. Our version relies on texture maps in many places where the original version used procedural texturing. The bowling pin shader computes texture coordinates given uv (the 2D surface parameterization), and the built-in variable Pobj (the object-space position). After being transformed by a set of transformation matrices, the texture coordinates are used to index texture maps specified by texrefs, which correspond to numeric OpenGL texture names. An alpha mask computed at the vertices is used to isolate some of the textures to either the front or back half of the bowling pin. Lighting is computed by two functions defined in an include file, one of which is described further in Section 5.3. Finally, we compute the final color by compositing the textures and applying the lighting. We rely on a feature in our language (described in Section 5.4) that allows us to control the computation frequencies of values and operations without specifying them explicitly. Note that this version of the bowling shader omits bump mapping and is therefore different from the versions of the shader used in the results section and video tape.

5 Shading Language

5.1 Language overview

The language we implemented is based to a loose degree on RenderMan. Several differences are noteworthy.

First, we decided to omit features not currently supported by mainstream graphics hardware. In particular, we omit support for data-dependent loops and conditionals. Adding support for this feature would require substantial changes to our language and compiler. While we omit support for data-dependent loops and conditionals, we designed our language and compiler to make support for new and less-restricted operators easy to add. Operations such as vertex textures and dependent texture lookups are already supported in our language; support for these operations has been disabled pending the necessary hardware support.

Second, we omit a number of features we felt were not essential to exploring the compilation and architectural issues we wanted to research. Features in this second category include atmosphere, imaging, and transformation shaders and a complete library of built-in functions (we provide only basic built-in functions). Over time we expect to enhance the language by adding some of these features.

Third, we deviated from RenderMan's syntax to reflect terms and techniques used by real-time graphics APIs such as OpenGL and to some extent Direct3D. Examples of syntactical changes that made it easier to develop shaders in the OpenGL environment include:

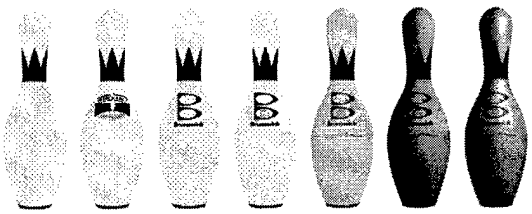


Figure 5: Constructing the bowling pin. We show the seven compositing steps use to compute the final color of the bowling pin shader in Figure 4. The images depict (a) Base texture; (b) after applying Coated (back half of pin shown), (c) Bruns, (d) Circle, and (e) Marks; (f) after multiplying by Cd; (g) after adding Cs. The images do not correspond to rendering passes, since more than one of these steps may be performed in a single pass.

- Types that reflect OpenGL vertex types, including RGBA colors and positions encoded in 4-vectors. RenderMan has only a single number representation, a float, but we introduced clamped floats to allow for numbers to be represented using fixed-point.
- Predefined vertex parameters that correspond to positions, normals, and other standard OpenGL parameters, plus support for tangent and binormal vectors. We also include a number of primitive-group parameters for the modelview and projection matrices and the light position and orientation.
- Textures denoted by texture references, or *texrefs*, rather than by strings. Texture formats reflect formats available in OpenGL, and these differ from the RenderMan formats.

To illustrate our shading language, we show an example surface shader in Figure 4.

Three more differences are important. First, we paid particular attention to the semantics of the language in order to support a high degree of optimization. Second, light shaders, light variables, and the combining of surfaces and lights are all handled differently from RenderMan. Finally, we split RenderMan's varying type into two separate types: vertex and fragment. We discuss these differences in the following sections.

5.2 Language analysis

An important property of our language that distinguishes it from the previous work is that our language is easily analyzed and optimized by a compiler. Analysis is important because it allows us to infer several kinds of information that users would otherwise have to specify explicitly. Optimization is particularly important in a real-time context for making shaders run as fast as possible.

Four aspects of the language help make it easy to analyze and optimize:

- **Function inlining.** We explicitly inline all functions and delay the analysis and optimization of each inlined function until after the function has been inlined. This allows the compiler to specialize each function to its calling context.
- **Combined compilation of surfaces and lights.** We compile surfaces and lights together, and we delay analysis and optimization until after surfaces and lights have been combined. This allows us to perform analysis and optimization across surface and light shaders.
- **No data-dependent loops and branches.** The lack of support for data-dependent loops and branches in hardware means we do not support these features in our language. This considerably simplifies the analyses we must perform.
- **No random access to memory.** The lack of hardware support for random read/write access to memory likewise allows us to eliminate that feature from our language. In particular, this

removes the possibility of pointer aliasing, again simplifying the analyses we must perform.

Together, *these properties of the language allow us to reduce shading calculations to a single DAG representing a complete pipeline program.* Once in this format, analysis and optimization are very straightforward.

5.3 Surface and light shaders

We support two shader types: surfaces and lights. Surface shaders return a 4-component RGBA color to be composited with the framebuffer, while light shaders return a 4-component RGBA light color to be used by surface shaders.

Compared to RenderMan, we use a slightly different syntax to combine lights with surfaces. We introduce a special *linear* integrate operator, which evaluates a "per-light" expression once for each light and sums the results. A simple integrate example:

```
// A lighting model for combined ambient and diffuse illumination
surface float4 lightmodel_diffuse (float4 ka, float4 kd) {
    perlight float NdotL = max(0, dot(N,L));
    return ka * Ca + integrate(kd * NdotL * Cl);
}
```

Note the per-light variable NdotL in the example. Our system defines three per-light values: the light color (Cl), the light vector (L), and the half-angle vector (H). If a surface shader uses one of these values, then dependent values must be computed once per-light. Our compiler infers which expressions in a surface shader are per-light by performing type analysis on expressions; however, to make code more readable, we require variables that hold per-light values to be declared with the perlight type modifier.

The integrate operator is converted into a sum at compile time. The compiler expands integrated expressions by replicating them for all the active lights, then summing the results. In the example above, the special per-light global Cl is replaced by the corresponding light shader's return value. When we build the integrate expression, we sort terms by computation frequency, grouping lights that return vertex values together. This allows multiple per-vertex light values to be added together in the vertex stage so that only a single per-vertex value is interpolated and added to the remaining light values in the fragment stage.

The integrate operator is linear in the sense that integrate(ka * a + kb * b) is equivalent to ka * integrate(a) + kb * integrate(b) if neither ka nor kb is per-light. The linearity of the integrate operator guarantees that certain optimizations can be made.

It is interesting to note that, for practical purposes, the programmable portions of the OpenGL and Direct3D APIs have lost the notion of separate surfaces and lights even though their non-programmable counterparts support the feature. Our shading language returns this feature to users of programmable hardware.

5.4 Support for computation frequencies

In Section 4, we introduced the concept of multiple computation frequencies. In our language, we represent multiple computation frequencies using four type modifiers: constant, primitive group, vertex, and fragment. We originally considered using RenderMan's uniform and varying type system, but chose not to once we realized that uniform and varying could not adequately represent the four computation frequencies we had identified. We introduce the new type modifier terminology to generalize the concept inspired by RenderMan's type system.

While our language contains type modifiers for computation frequencies, the user may choose to specify values with or without the modifiers. If a value is specified with a computation frequency, then it will be forced to have the computation frequency that was specified. If a value is specified without a computation frequency, our compiler applies a set of rules to infer the appropriate computation frequency. These inference rules allow users to manage computation frequencies without the hassle of explicit specification.

Initially, we considered designing our language to require explicit specification of all computation frequencies; however, we soon realized this would be very inconvenient for the user. Aside from being tedious, explicit specification of computation frequencies makes it difficult to write a surface shader that efficiently handles both vertex and fragment lights. With computation frequencies explicitly specified, a surface shader written to accommodate fragment lights will not perform efficiently if only vertex lights are used. The inference mechanism allows the user to leave the computation frequencies of surface shader computations unspecified so that those computations may be optimized in a way that accounts for the computation frequencies of the active lights. This in turn results in significant computational savings.

Two rules are used to infer computation frequencies. The first deals with the default computation frequencies of shader parameters, while the second deals with the propagation of computation frequencies across operators. Given these two rules, the compiler can always infer the computation frequency of a given value or operator by propagating computation frequencies from shader parameters forward.

All shader parameters have a default computation frequency. The default computation frequency depends on the parameter's type and the class of the corresponding shader (surface or light). For example, floating-point scalars and vectors default to vertex for surface shaders and to primitive group for light shaders; matrices default to primitive group for both kinds of shaders:

```
surface shader float4 surf1 (float f) { ... }      // f is vertex
light shader float4 light1 (float f) { ... }       // f is primitive group
```

Default computation frequencies may be overridden if the user specifies the computation frequency of a parameter explicitly:

```
light shader float4 light2 (vertex float f) { ... }    // f is vertex
```

The computation frequencies of computed values are determined by applying a second rule that propagates computation frequencies across operators. In general, the second rule attempts to minimize total computation by performing computations at the least-frequent computation frequency possible. Because it is impossible to demote a value from a more-frequent computation frequency to a less-frequent one, when combining values of different computation frequencies, the result varies as often as the most-frequent input operand. For example, adding a vertex value to a fragment value results in a fragment value. The second rule also obeys a number of additional computation frequency constraints for special operations (such as texturing) to satisfy the limitations of those operations.

While the computation frequencies of computed values are inferred using the rules just described, they may be controlled by type-casting values to specific computation frequencies. For example, if two vertex values N and L are to be used to compute dot(N,L), the result of the dot product will normally be per-vertex. However, a per-fragment dot product can be achieved by first casting N or L (or both) to a fragment value, e.g.:

```
dot((fragment float3)N, (fragment float3)L)      // dot is fragment
```

Note that the rules for inferring computation frequencies do not provide compilers any flexibility with respect to selecting computation frequencies. Users can always predict the computation frequencies that will be inferred, and therefore users always have full control over computation frequencies.

The process of propagating computation frequencies to operators and values labels each operator with a computation frequency. Our compiler uses this computation frequency information to assign operations to particular stages of the programmable pipeline.

6 Retargetable Compiler Back End

In this section we describe our retargetable compiler back end, which implements the programmable pipeline abstraction by mapping pipeline programs to shader object code. We designed our back end with two goals in mind: to support a wide variety of hardware and to support arbitrarily-complex computations.

To support a wide variety of hardware, we implement a modular compiler. New hardware can be targeted simply by adding new modules. We provide for separate modules for each computation frequency to allow modules to be interchanged and to allow for sharing of certain common modules.

Each module implements a single stage of the programmable pipeline and has two parts: a compile-time part and a render-time part. The compile-time part is necessary to target computations to specific hardware, while the render-time part is necessary to configure and utilize that hardware during rendering.

In all, we implement seven back-end modules. We implement two primitive group back ends (cc-pg and x86-pg), both of which target host processors. We also implement three vertex back ends, two for the host processor (cc-v and x86-v) and one for programmable vertex-processing hardware (vprog). We also implement two fragment back ends, one for the standard OpenGL pipeline plus a number of optional extensions (lburg), and one for programmable fragment-processing hardware (regcomb).

We use two techniques to support arbitrarily-complex computations. First, we use multipass methods if a single hardware pass is unable to execute the entire fragment portion of a pipeline program. Second, we fall back to host processing for vertex computations if the available vertex-processing resources are insufficient.

6.1 Module interactions

In the following sections, we describe individual modules in detail. However, one of the major complexities in the system is that modules are not completely independent. We now discuss three important kinds of interactions and some of our implementation strategies for dealing with them:

Data flow interactions. Data values must flow from the user application into the shading system and through the stages of the programmable pipeline. For modules to interact properly, we must define the format of the data that is passed between stages. All values computed or specified on the host are stored in a fixed format that is the same for all back ends. Values that are computed on a vertex or fragment processor use a format specific to that processor, since they must be communicated to the following stage.

As an example, consider passing vertex values from the host CPU to the graphics processor. With non-programmable vertex-processing hardware, we use the host to perform the necessary vertex computations, and we pass computed vertex values to the hardware. With programmable vertex-processing hardware, we pass user-specified vertex parameters directly to hardware. To facilitate the efficient passing of both kinds of vertex values, we format all vertex data using vertex arrays.

Pass-related interactions. The fragment back ends may rely on multiple passes to implement arbitrarily-complex fragment computations. A complication occurs when using multiple passes with programmable vertex-processing hardware: we must partition vertex computations according to which values are needed by each pass. To handle this case, fragment back ends compile their code first, then provide lists of values to vertex back ends to indicate which values are needed for each rendering pass.

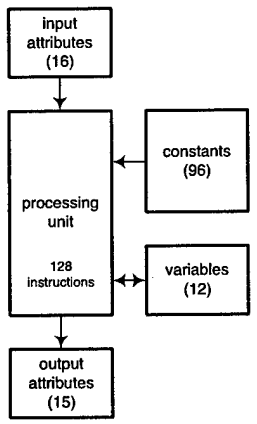
Resource constraint interactions. When using the programmable vertex-processing hardware back end, it is possible for a fragment back end to request a set of values for a particular pass that cannot be computed given the available vertex-processing resources, such as registers and instructions. To allow our system to handle this case, we rely on the modularity of our system and fall back to one of the host-side vertex back ends. More sophisticated solutions are possible, such as negotiating simpler passes with the fragment back end, but we do not attempt any of them.

6.2 Host-side back ends

We implement four host-side back ends, two of which support primitive-group computations and two of which support vertex computations. We initially implemented these back ends because they offered us a convenient way to explore primitive-group and vertex programmability. However, we continue to use all four back ends and consider them to be an important part of the system. The primitive-group back ends are useful because current hardware does not support the primitive-group functionality we require. The vertex back ends are useful because they allow for vertex programmability when programmable vertex hardware is unavailable.

All four host-side back ends generate code by traversing the internal representation and emitting code templates. Two of the back ends use a common set of routines to emit C code, generate a dynamically-linked library using an external C compiler, and load the compiled shader code. Likewise, the other two back ends use a common set of routines to emit x86 assembly and generate x86 object code internally. We found the C compiler approach to be very portable, and we note this approach generates better code than the internal x86 code-generation approach; however, we prefer the internal x86 code-generation approach because it generates code quickly and without the hassle of a properly-configured external compiler.

6.3 Vertex program back end




```
// Output surface position in clip-space
DP4 o[HPOS].x, v[OPOS], c[0];
DP4 o[HPOS].y, v[OPOS], c[1];
DP4 o[HPOS].z, v[OPOS], c[2];
DP4 o[HPOS].w, v[OPOS], c[3];
// Transform normal to eye-space
DP3 R0.x, v[NRML], c[4];
DP3 R0.y, v[NRML], c[5];
DP3 R0.z, v[NRML], c[6];
// Normalize eye-space normal
DP3 R0.w, R0, R0;
RSQ R0.w, R0.w;
MUL R0, R0, R0.w;
// Compute N · L and clamp using LIT
DP3 R0.x, R0, c[7];
LIT R0, R0;
// Multiply by diffuse color and output
MUL o[COL0].xyz, R0.y, c[8];
```

Figure 6: Vertex program architecture. The vertex program architecture processes a single vertex at a time and computes a set of output attributes given a number of input attributes, constants, and variables (i.e. temporary registers). To give an impression of the architecture's programming model, a sample program that computes a diffuse lighting term given an infinite light is shown. In the example, c[0..3] are the rows of the composite matrix, c[4..6] are the rows of the inverse modelview matrix, c[7] is the light direction, and c[8] is the diffuse color.

NVIDIA and Microsoft have recently proposed a vertex program architecture, shown in Figure 6. The architecture defines a register machine that conceptually operates on one vertex at a time. A processing unit with a capacity for 128 instructions computes up to 15 output vertex attributes given 16 read-only input attributes, 96 read-only constants (shared across all vertices), and 12 read-write variables (i.e. temporary registers). The machine is optimized to

process 4-vectors, and therefore most data paths are 4 wide. The instruction set contains 15 instructions including a number of basic but flexible instructions plus two specialized instructions, LIT and DST, which are used to accelerate lighting and attenuation computations. The architecture contains swizzle-on-read, negate-on-read, and write-masking features to facilitate the efficient processing of scalar and vector values. The architecture also has limited support for primitive-group computations, but we do not make use of this functionality because it is not flexible enough for our needs. Further information about the architecture can be found in [10, 16].

The vertex program back end maps computations to the programmable vertex-processing architecture just described. As with the host-side back ends, we generate instructions by traversing the internal representation and emitting code templates; however, unlike the host-side back ends, we perform this operation once per pass. For each pass, we generate code to compute the vertex values needed by the pass. (Note that some computations, position computations in particular, are needed by multiple passes and are therefore repeated across passes as necessary.) Instructions in code templates reference an infinitely-sized set of scalar and vector registers. After all instructions for a pass have been emitted, we perform register allocation to map this infinite register set to actual hardware registers.

We apply two general kinds of techniques to optimize instruction usage: code transformations, which occur before instruction selection, and peephole optimizations, which occur after instruction selection. Both help to reduce the number of instructions to help us stay within the 128 instruction limit. Some of the optimizations we implement include:

- collapse MUL and ADD to MAD (multiply-and-add)
- perform copy propagation of various sorts
- replace simple negations with negated source operands
- group parallel scalar operations into a single vector operation with output write-masking if necessary
- transform certain patterns of conditionals, clamps, and power operations to use the LIT instruction
- transform certain patterns which compute attenuation factors to use the DST instruction

Intermediate values are stored in variable registers. To optimize variable register usage, we order instructions according to a depth-first traversal, then apply a standard greedy graph-coloring register-allocation algorithm. While the depth-first traversal is not optimal, it helps to reduce the number of registers needed to store intermediate results. When graph coloring, we treat scalars as if they occupy a full vector register. We found that because we almost always have an adequate number of variable registers, this approximation works reasonably well. Note also that graph coloring is simplified because we cannot spill registers.

Constant and primitive-group values used by the vertex stage are stored in constant registers. Each primitive-group value is assigned its own constant register. Constant values, which are known at compile-time, are packed together, using the architecture's swizzle-on-read and negate-on-read functionality to extract actual constant values. For example, the scalars 0, 1, and -0.5, plus the vectors {.707, 0, .707, .5} and {0, −.707, −.707, -1} can all be packed into a single 4-component constant register as $\{.707, 0, .5, 1\}$. The constant packing algorithm first sorts constants in descending rank order, where the rank of a constant is the number of unique components it has. (For example, the rank of the vector {.707, 0, .707, .5} is three.) The algorithm then assigns each constant to a register, trying to minimize the impact of each constant by searching for matches with registers that have already been filled. Constant packing is important because a single program can access a large number of constants that share common values (this is especially true for matrices) and because it allows constants from consecutive passes to be packed together (although we do not perform this second optimization).

6.4 Generic lburg-based fragment back end

Our first of two fragment back ends compiles fragment computations to the OpenGL pipeline using multipass rendering techniques described by Peercy et al. We treat the OpenGL pipeline as implementing two basic kinds of passes: a render pass which computes a value into the framebuffer and a save pass which copies framebuffer memory to texture memory. Two equations summarize the two kinds of passes:

$$\begin{aligned} FB &= \{C, T, C \odot T\}[\odot T][\odot FB] && \text{(render)} \\ T &= FB && \text{(save)} \end{aligned}$$

C is a constant or interpolated color, T is the result of a texture lookup, FB is the framebuffer, and each $\odot$ is one of add, subtract, multiply, or blend. We use {...} to indicate "one of ..." and [...] to indicate that "..." is optional, so valid render passes include C, $T \odot FB$, and $C \odot T \odot T \odot FB$ among others. Render passes may also contain canned functions for bump mapping (described in Section 4.3), but for simplicity we omit these variations from the equations above.

We map DAGs of fragment computations to render and save passes using a bottom-up tree-matching technique similar to that used by Peercy et al. Specifically, we decompose the input DAG into trees, then we use a tree-matcher generated by *lburg* [4] to select a minimal-cost set of passes. Tree-matching is based on a set of rules derived directly from the render and save equations above.

We assign a cost of one to each render pass and a cost of five to each save pass. The difference in costs tends to eliminate unnecessary save passes, which are almost always slower than render passes. Also, because each render pass has the same cost, more operations tend to get packed into fewer passes.

We implement save passes by copying the entire framebuffer to texture. This is in contrast Peercy et al. who only copied the bounding box of the objects being rendered. Bounding box information is not readily available to us, so we do not use it. Given render-to-texture functionality, we could have eliminated the copies altogether, but this functionality is not yet available in OpenGL.

Peercy et al. proposed the use of tree-matching algorithms to target OpenGL extensions such as multitexture. We provide support for multitexture, including a few of the simpler texture combining extensions. We handle extensions by using the lburg cost mechanism to dynamically enable and disable rules that depend on the availability of certain extensions. A very large cost, set at run time when an extension is found to be missing, effectively disables a rule.

We found the tree-matching technique just described to be effective when passes are simple; however, when we attempted to extend the technique to more-complex, programmable fragment hardware, we encountered a number of difficulties:

- **Resource management.** An important aspect of targeting programmable hardware is allocating and managing resources such as instructions, registers, constants, interpolants, and textures, all of which are available in limited amounts within a single pass. The tree-matching technique has no way to track these resources. In addition, recent combiner architectures support independent RGB and ALPHA operations, and tree-matching has difficulty managing functional units that can be separated in this manner.
- **Handling of DAGs.** The tree-matching algorithm matches trees, not DAGs. Our lburg back end handles values that are referenced more than once either by splitting them off into a separate tree or by duplicating them and recomputing them

once for every use. Values that are split off are saved to texture memory. Since this operation is expensive, we prefer to duplicate rather than to split; however, we only duplicate values that match a set of patterns we know fit into a single pass.

Decomposing DAGs into trees for tree-matching adds rendering overhead. If a single rendering pass is simple enough that it can only evaluate a tree of operations, then the overhead is minimal, since any pass-selection algorithm will ultimately generate a similar decomposition. However, if a single rendering pass can evaluate a DAG of operations, as is typically the case with programmable fragment hardware, then decomposition may not be necessary and overhead costs may be realized.

- **Tree permutations.** Our tree-matching algorithm uses a hierarchical set of rules to define tree patterns to be matched. Through the use of registers, programmable hardware is able to express a very large number of tree patterns. Assuming instructions with two inputs, the number of rules needed to express all possible patterns grows as the square of the number of instructions available, which quickly becomes unmanageable. The situation is much worse if instructions have more than two inputs.

These difficulties convinced us to abandon our attempts to use pass-based tree-matching techniques to target programmable fragment hardware.

6.5 Programmable fragment hardware back end

To address the problems of the tree-matching technique, we developed a second fragment back end specifically designed to target programmable fragment hardware. This back end currently targets the NV_register_combiners OpenGL extension, but could be easily modified to target the DirectX 8 pixel-shader instruction set, which exposes similar hardware functionality.

The register combiner architecture, like the vertex program architecture, is register-based. Conceptually, it operates on one fragment at a time. A processing unit called a register combiner operates on a set of registers and constants to compute new values, which are then written back into registers. Registers are initially loaded with interpolated vertex colors and the results of texture lookups. The architecture allows the number of registers and textures to vary with degree of multitexture supported. The number of register combiner units is also allowed to vary.

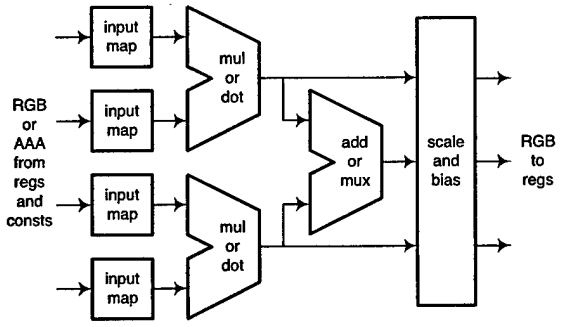


Figure 7: RGB register combiner architecture. An RGB register combiner processes four inputs to compute three outputs. The combiner computes two product terms and a sum/mux term. It can apply an input mapping to each input, and can scale and bias its outputs given a single shared scale/bias factor.

A register combiner consists of two parts: an RGB portion and an ALPHA portion. The RGB portion of a register combiner is depicted in Figure 7. It consists of four 3-component inputs and three 3-component outputs. Each input comes from either the RGB portion of a register or the ALPHA portion of a register replicated across all three components. One of eight mappings may be applied to each input to scale, negate, and/or bias the input. Three values are then computed from the four mapped inputs. Two of the values are computed as either the product or the dot product of an associated pair of inputs. The third value is either the sum or a mux of the first two values, with the restriction that if either of the first two operations was a dot product, the third value must be discarded. Before being written to registers, all three values are scaled and biased by a single shared scale/bias factor. The ALPHA combiner is similar to the RGB combiner except that the ALPHA combiner has scalar inputs and outputs, where each input comes from either the ALPHA or the BLUE portion of a register. Because the ALPHA combiner operates on scalar values, it does not perform dot products.

The register combiner architecture also specifies a final combiner designed to perform a fog computation but capable of performing other computations also. Details can be found in [16].

We target the register combiner architecture using a compilation algorithm that treats the architecture as if it were a VLIW processor, with register combiners corresponding to VLIW instructions. The complete details of our compilation algorithm are beyond the scope of this paper and are described in a separate publication [11]. We outline our basic approach here.

The core of our algorithm maps a DAG of operations to a single rendering pass in five steps:

1. **Rewrite input DAG.** We first preprocess the input DAG to split RGB and ALPHA computations, to expand certain index operations using dot products, and to expand select operations to use the architecture's less-than-half muxing scheme.
2. **Determine input and output scale/bias mapping operations.** We scan the input DAG for sequences of operations that correspond to mapping operations and replace each sequence with a single mapping operation. We perform range analysis to find situations where mappings intended for [0,1] numbers can be applied to numbers that are [–1,1] by type but [0,1] after analysis.
3. **Select instructions.** We perform a greedy, top-down DAG traversal to map the input DAG to register combiners. We assign operations to half combiners (product only) and whole combiners (sum of products) and we select RGB and ALPHA combiners as appropriate to the computations being performed. We currently only use the final combiner in a limited fashion. The output of this step is a DAG of register combiner instructions.
4. **Allocate pass inputs.** We use a greedy algorithm to map pass inputs to their initial registers. We are especially careful to allow as many paths as possible for values of various types (constant/interpolated scalar/vector color values plus textures) to be allocated to registers.
5. **Schedule instructions and allocate registers.** We sort the instructions in the instruction DAG by decreasing node depth, then greedily schedule instructions to the first appropriate slot available. As we schedule instructions, we also reserve register space for results; we free register space on the last use of a result. We make a special effort to properly manage the alpha component of the SPARE0 register, which has exclusive control over the MUX operation.

Our implementation does not yet decompose DAGs larger than a single pass into pass-sized pieces for scheduling by our core single-pass algorithm. While it is clear to us that it would be straightforward to generate a correct decomposition of any input DAG, it remains to be seen how efficient we can make these decompositions.

The most difficult aspect of compiling to register combiners is dealing with idiosyncrasies in the architecture. In particular, many aspects of the architecture are not orthogonal. For example, the sharing of a single output scale/bias factor by all three combiner outputs complicates both instruction selection and instruction scheduling, and the requirement that the MUX operation's control input come from the alpha component of the SPARE0 register complicates both instruction scheduling and register allocation.

A more fundamental problem with the register-combiner architecture is the wide variety of fixed-point data types it uses. Values stored in registers have a range of [–1,1], but intermediate results within a single combiner can have other ranges, such as [–2,2] and [0,4]. Ideally, a shading language has well-defined range semantics for its data types, but because register combiner operations and data types are not orthogonal, register combiners do not cleanly support this ideal. A [–1,1] type with proper semantics can be implemented, but only with a performance penalty. We forgo the ideal, implicitly exposing the hardware's range semantics in the language. When needed, the user may explicitly request [–1,1] clamping. Ultimately, we hope this problem will be fixed in hardware with the addition of consistent and orthogonal support for a small set of well-defined data types.

We anticipate that future hardware will support more registers, more textures, and more combiners than current hardware. To accommodate such changes, we designed our programmable fragment back end to compile to a parameterized model of hardware. We also designed our system to facilitate the addition of support for NV_texture_shader texture-addressing operations [16].

7 Results

Several of the results in the following sections are shown in our video on the SIGGRAPH 2001 Conference Proceedings video tape.

7.1 Shading language

Vertex vs. fragment tradeoff. Our language allows us to easily express many computations using either vertex or fragment processing. To demonstrate this, we coded up two versions of the Banks anisotropic reflection model [2], one version based on Heidrich's algorithm [6], with the lighting model stored in a texture and texture coordinates computed at vertices, and a second version with the entire lighting model computed at each vertex.

The tradeoffs are evident as the surface dicing and number of lights are changed. For a simple spherical surface, the vertex-based algorithm requires a dicing factor around 3 to 4 times higher in each dimension for quality equivalent to that of the textured version, while the textured version requires an additional texture lookup per light. Using our lburg back end, the textured version requires one additional pass per light. No additional passes are needed for the vertex-based algorithm.

Combined compilation of surface and light shaders. In our language, we compile surface and light shaders together and delay the optimization of the surface shader until after the shaders have been combined. This significantly enhances the ability of our compiler to optimize computations.

Combined compilation allows us to specialize surface shader code to match the computation frequencies of the active lights. Without combined compilation, we would have to compile surface shaders assuming the worst case: that all lights are fragment lights. This would in turn cause vertex lights to be handled inefficiently. We can illustrate the savings by examining a simple surface shader that computes integrate(fr * Cl), the sum of products of per-vertex reflection factors fr and light colors Cl. If all lights are vertex lights, combined compilation allows us to recognize that the sum of products may be performed per-vertex. In this case, the example would run in one pass regardless of the number of lights. However, without combined compilation, the sum of products would have to be performed per-fragment, requiring one fragment multiply for the first light plus one fragment multiply and one fragment add for each additional light. With our lburg back end, the example would then require two render passes for the first light plus two render passes and one save for each additional light.

Combined compilation also allows us to sort lights by computation frequency to minimize the portion of the light sum that must be performed per-fragment. Given the previous example surface, two simple vertex lights, and one simple fragment light, the sorting optimization allows our lburg back end to compile the shaders to three render passes. Without the sorting optimization, the worst-case ordering of lights would cause our lburg back end to generate six render passes and two save passes.

7.2 Vertex-program back end

To assess the efficiency of our compiler's vertex-program back end, we compared the output of our compiler with a hand-written vertex-program that performs the same computation. For the comparison, we used a surface/light shader pair that computes a per-vertex color using a variant of the OpenGL shading model. A specular/diffuse/ambient reflection is computed using a local light with a quadratic distance attenuation factor. We use a local eye point.

Our compiler-generated vertex program uses 44 instructions. We created the corresponding hand-written vertex program by selecting and optimizing pieces from an NVIDIA template [9]. The hand-written program uses 38 instructions. The six extra instructions of the compiler-generated program fall into two categories. Four instructions result from sub-optimal code generation. The other two are required to support both local and infinite lights, because our system doesn't currently provide any means to specify at compile time whether a light shader will be used with local lights ($L_w \neq 0$), or directional lights ($L_w = 0$).

This example, and our broader experience with the system, demonstrate that the performance of vertex computations expressed in a high-level language can approach the performance of hand-written assembly code.

7.3 Fragment back ends

To evaluate the efficiency of our register-combiner back end, we implemented a per-fragment shading model that uses a variant of Kilgard's hardware-friendly bump-mapping algorithm [8]. The shading model includes specular, diffuse, and ambient terms and is expressed in our shading language directly, i.e. without using the canned bump-mapping functions. Textures are used to store normals and spatially-varying diffuse/specular reflection coefficients. The shading model uses 14 three-vector operations and 12 scalar operations, including the operations required by the bump-mapping algorithm. Broken down by operation type, the shading model uses 14 multiplies, 6 adds, 3 clamps, 2 dot products, and 1 select. It also uses four texture lookups.

Our register-combiner back end compiles this shading computation to a single pass on an NVIDIA GeForce3, using four texture units and seven register combiners. Using the compiler output as a guide, we were able to tune the source code to reduce the combiner count to five. We were unable to do any better by hand coding the shader, although hand coding did allow the "final" register combiner to be used in place of one of the "standard" combiners, which might improve performance on some hardware.

In contrast, our lburg back end requires six rendering passes plus a framebuffer-to-texture copy for the same shading model. In order to achieve this performance, we had to replace the bump-mapping

code with the built-in canned functions that invoke hand-written register-combiner code for the diffuse and specular bump-map computations.

The following is a summary of our initial experiences with our register-combiner compiler:

- We have yet to find a shader which runs out of fragment registers before other resources, even though our compiler's instruction-scheduling algorithm is actually biased towards heavy register usage.
- When compiling to a GeForce2 (which has two combiners and two texture units), we found that we usually run out of either textures or instructions first, depending on the type of shader.
- When compiling to a GeForce3 (which has eight combiners and four texture units), we usually run out of textures or interpolators before we run out of instructions, although some shaders run out of instructions first.
- Scalar computations are surprisingly frequent. The compiler can place scalar computations in RGB combiners, and this optimization has proved to be important for scalar-heavy shaders.

7.4 System Demonstration

To demonstrate the full capabilities of our system, we constructed two example scenes. We ran both scenes at a resolution of 640x512 on an 866 MHz Pentium III system with an NVIDIA GeForce3, which supports the NV_vertex_program and 8-combiner NV_register_combiners extensions.

Textbook strike. We implemented a version of the textbook strike scene from the cover of [23] using animation data provided by UNC. Our version has ten bump-mapped bowling pins and their bump-mapped reflections, plus a bowling ball and a textured floor. The scene contains a total of four surface shaders and one light shader. We optimized the bowling-pin shader to compile to one pass with our register-combiner back end by pre-compositing the three projective decal textures into a single projective decal texture. We are able to run this animation at 55 frames/sec. A single frame of our animation is shown in Figure 8.

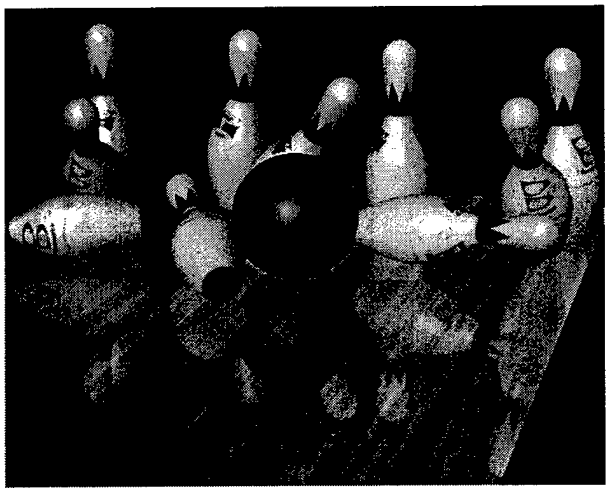


Figure 8: Textbook strike. This scene contains ten bump-mapped bowling pins and their bump-mapped reflections, plus a bowling ball and a textured floor. It runs at 55 frames/sec on a GeForce3.

Fish. We also implemented a swimming fish scene using animated fish data originally from [22]. The scene contains a fish with a bump-mapped body, transparent bump-mapped fins, a textured ground plane, a fragment light casting a caustic texture on all objects, and a ground-plane shadow for the fish. In total, there are five surface shaders and one light shader. For this scene, we use our lburg back end to compile all of the shaders, and we also use our immediate-mode interface to specify all geometry. We are able to run this animation interactively at 22 frames/sec. A single frame of animation is shown in Figure 9.

Both of these scenes run on a wide range of hardware from different vendors. When necessary, shaders use conditional compilation to provide fallback paths when hardware bump mapping isn't available. The scenes run on a wide variety of hardware, including basic OpenGL hardware from SGI, multitexturing hardware from 3dfx and ATI, and programmable hardware from NVIDIA. In addition, our system adapts the scenes to different generations of hardware from the same vendor, taking advantage of features in each chipset.

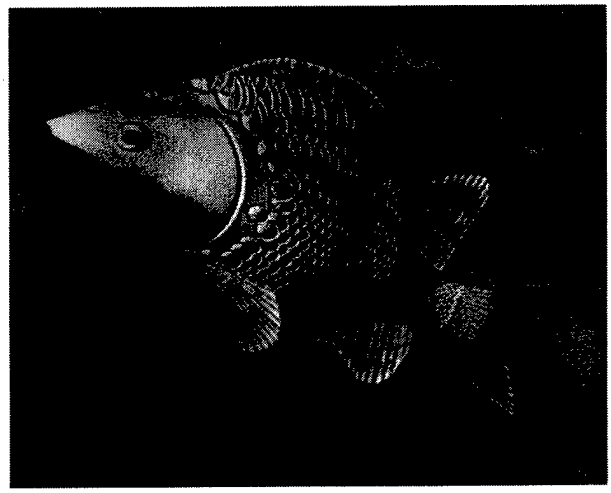


Figure 9: Fish. This scene contains a fish with a bump-mapped body and transparent, bump-mapped fins plus a fragment light that casts a caustic texture on all objects. It runs at 22 frames/sec on a GeForce3.

8 Discussion and Future Work

In this paper, we described the implementation of a real-time procedural shading system designed for programmable graphics hardware. We introduced a programmable pipeline abstraction to unify the notion of multiple computation frequencies and to support pipeline virtualization. We described a shading language tailored to graphics hardware and introduced new schemes for optimizing computations, combining surfaces and lights, and managing computation frequencies. We also described our retargetable implementation of the programmable pipeline using modules that target current graphics hardware, including support for programmable vertex and fragment hardware. Finally, we demonstrated our system running in real-time on today's graphics hardware.

It is much easier to program in our language than it is to write custom multipass OpenGL programs. Furthermore, shaders written in our language are portable, since the compiler handles the details of mapping shaders to graphics architectures with different features. The language itself is simpler and less ambitious than RenderMan. Although we could wait until the graphics hardware is fast enough to completely implement the RenderMan shading language, we think—given the current capabilities and rapid advances in graphics hardware—that a better strategy is to demonstrate its feasibility now, then allow our system to evolve over time. It should be noted that in order for developers of real-time rendering applications such as games to adopt shading languages, it is most important that the

languages be compiled to the hardware near optimally. Having lots of features is less critical.

A number of hardware improvements would help with the implementation of our programmable pipeline abstraction. The first is support for orthogonality of operations and data types across computation frequencies, including vertex textures as well as fragment floating-point and full support for dependent texturing as described by Peercy. It is difficult to compile to fragment register combiners given their current restrictions and special cases. Second, virtualizing the hardware using multiple passes requires the ability to spill intermediate fragment values to the framebuffer. Currently, high precision intermediate results cannot be stored in the framebuffer, making it difficult to split a computation into multiple passes. Third, support for rendering transparent geometry currently requires us to separately render potentially-overlapping transparent objects. This remains a big limitation of multipass rendering and could be fixed with changes to hardware [12].

Currently, our programmable pipeline abstraction is just an internal interface for communicating shading computations between our system's front and back ends. We have demonstrated that the intermediate format may be mapped to a variety of different hardware architectures by our compiler. In a similar way, we would like to follow Peercy et al.'s suggestion and develop several domain-specific languages [19] and implement them as different front ends. A language intended for artists (as opposed to graphics programmers) might be particularly relevant to potential users of our system.

So far, our experiences with real-time procedural shading on graphics hardware have been very encouraging. In the microprocessor world, the instruction sets of microprocessors changed radically as appropriate compiler technology was developed. This in turn allowed innovative hardware designs that might not have been possible otherwise. In a similar way, we think it is possible to develop future graphics hardware optimized to run a programmable graphics pipeline.

9 Acknowledgements

First, we thank all the people at Stanford who have worked with us. The textbook strike and fish demos were written by Pradeep Sen and Ren Ng respectively. In addition, Pradeep added several optimization paths to our vertex program back end. John Owens and Bill Dally from the Stanford Imagine project have been working on a back end to target the Imagine processor in part to help us demonstrate the flexibility of our programmable pipeline abstraction. Philipp Slussalek was at Stanford when our project first started, and he participated in many of our early discussions. Many other people in and around the graphics lab at Stanford contributed their advice and support at various times.

Second, we thank the people who provided us with useful data and code. Anselmo Lastra, Lawrence Kesteloot, and Fredrik Fatemi provided us with position and orientation data for the textbook strike scene. Xiaoyuan Tu and Homan Igehy provided us with the fish animation data. Wolfgang Heidrich sent us code to implement his texture-based anisotropic reflection model.

Finally, we thank our project's sponsors: DARPA, ATI (Andy Grueber and Steve Morein), 3dfx (John Danskin, Roger Allen, and Gary Tarolli), SUN (Michael Deering), SGI (Mark Peercy and Marc Olano), Sony-Kihara Research Center (Takashi Totsuka and Yuji Yamaguchi), and NVIDIA (Matt Papakipos, Mark Kilgard, and David Kirk). Conversations with all of our sponsors have contributed significantly to our understanding of various hardware issues. We particularly thank Matt and Mark from NVIDIA for their feedback and their assistance with drivers and hardware.

References

[1] A. A. Apodaca and L. Gritz. *Advanced RenderMan: Creating CGI for Motion Pictures*. Morgan Kaufmann, 2000.

[2] D. Banks. Illumination in Diverse Codimensions. In *SIGGRAPH 94 Conference Proceedings*, pages 327–334, July 1994.

[3] R. L. Cook. Shade Trees. In *Computer Graphics (SIGGRAPH 84 Conference Proceedings)*, pages 223–231, July 1984.

[4] C. Fraser and D. Hanson. *A Retargetable C Compiler: Design and Implementation*. Addison-Wesley, 1995.

[5] P. Hanrahan and J. Lawson. A Language for Shading and Lighting Calculations. In *Computer Graphics (SIGGRAPH 90 Conference Proceedings)*, pages 289–298, Aug. 1990.

[6] W. Heidrich and H.-P. Seidel. Realistic, Hardware-accelerated Shading and Lighting. In *SIGGRAPH 99 Conference Proceedings*, pages 171–178, Aug. 1999.

[7] P. Jaquays and B. Hook. *Quake 3: Arena Shader Manual, Revision 10*, Sept. 1999.

[8] M. J. Kilgard. A Practical and Robust Bump-mapping Technique for Today's GPU's. Technical report, NVIDIA Corporation, July 2000. Available at http://www.nvidia.com/.

[9] E. Lindholm. Vertex Programs for Fixed Function Pipeline. NVIDIA Technical Presentation (from www.nvidia.com), Nov. 2000.

[10] E. Lindholm, M. J. Kilgard, and H. Moreton. A User-Programmable Vertex Engine. In *SIGGRAPH 01 Conference Proceedings*, Aug. 2001.

[11] W. R. Mark and K. Proudfoot. Compiling To a VLIW Fragment Pipeline. In *Eurographics/SIGGRAPH Workshop on Graphics Hardware*, Aug. 2001.

[12] W. R. Mark and K. Proudfoot. The F-Buffer: A Rasterization-Order FIFO Buffer for Multi-Pass Rendering. In *Eurographics/SIGGRAPH Workshop on Graphics Hardware*, Aug. 2001.

[13] M. D. McCool. SMASH: A Next-Generation API for Programmable Graphics Accelerators. Technical Report CS-2000-14, University of Waterloo, Aug. 2000.

[14] Microsoft. *DirectX 8.0 Programmer's Reference*, Oct. 2000.

[15] S. Molnar, J. Eyles, and J. Poulton. PixelFlow: High-Speed Rendering Using Image Composition. In *Computer Graphics (SIGGRAPH 92 Conference Proceedings)*, pages 231–240, July 1992.

[16] NVIDIA Corporation. *NVIDIA OpenGL Extension Specifications*, May 2001. http://www.nvidia.com/developer/.

[17] M. Olano. *A Programmable Pipeline for Graphics Hardware*. PhD thesis, University of North Carolina at Chapel Hill, 1998.

[18] M. Olano and A. Lastra. A Shading Language on Graphics Hardware: The PixelFlow Shading System. In *SIGGRAPH 98 Conference Proceedings*, pages 159–168, July 1998.

[19] M. S. Peercy, M. Olano, J. Airey, and P. J. Ungar. Interactive Multi-Pass Programmable Shading. In *SIGGRAPH 00 Conference Proceedings*, pages 425–432, July 2000.

[20] K. Perlin. An Image Synthesizer. In *Computer Graphics (SIGGRAPH 85 Conference Proceedings)*, pages 287–296, July 1985.

[21] M. Segal, K. Akeley, C. Frazier, and J. Leech. *The OpenGL Graphics System: A Specification (Version 1.2)*, Mar. 1998.

[22] X. Tu and D. Terzopoulos. Artificial Fishes: Physics, Locomotion, Perception, Behavior. In *SIGGRAPH 94 Conference Proceedings*, pages 43–50, July 1994.

[23] S. Upstill. *The RenderMan Companion: A Programmer's Guide to Realistic Computer Graphics*. Addison-Wesley, 1990.